Week 6

Cryptographic Hashes and Digital Signatures

THE UNIVERSITY OF SYDNEY

Dr. Thilini Dahanayaka

School of Computer Science, The University of Sydney

Agenda

- Cryptographic hash functions
- Requirements and Security
- Secure Hash Algorithm (SHA)
- Message Authentication Code (MAC)
- Digital Signatures

Recommended Reading

- Cryptography and Network Security (7th Edition - William Stallings):
  - **Chapter 11** - Cryptographic Hash Functions
  - **Chapter 12** - Message Authentication Codes
  - **Chapter 13** - Digital Signatures

1. Cryptographic Hash Functions

Hash Functions

- **Goal:** Integrity
- We want to ascertain that a received message is the same as the one that was sent
- **Idea:** define a function to create a checksum (‘hash function’)
  - Sender applies hash function to message and obtains a checksum (‘hash value’ or just ‘hash’)
  - Sender encrypts message and sends together with hash value of plaintext
  - Receiver decrypts message, applies hash function, and compares with the transmitted hash value
  - If the hash values match, the plaintext must be correct
- **Immediate problems:** Adversary must not be able to retrieve or guess the plaintext from the hash value
- Other problems also exist, but let’s focus on this one for now

Common Hash Functions

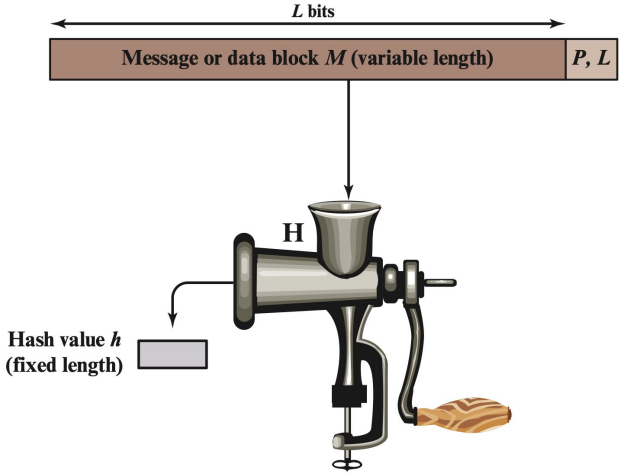
- **Hash Function (H)** A function that takes an input of variable length and produces a fixed-length output h = H(M)
- Also called: Digest.
- Examples:
  - CRC (CRC-16, CRC-32, …)
  - MD5 (no longer used) – SHA1 (in phase-out)
  - SHA-2, SHA-3
  - RIPEMD160
- Difference between CRC and the others: CRC is not designed for cryptography
- MD5 and SHA1 were designed for use in cryptography but were found to be insufficient

What are we trying to prevent?

- In order to communicate securely, we normally want both confidentiality and integrity.
- With hash functions, we want to prevent the following for integrity:
  - Attacker can somehow get the plaintext from the hash value alone (does not defeat integrity, but defeats our encryption)
  - Attacker can find a bit sequence that has the same hash value (decryption of a bit sequence may yield implausible plaintext, but plausibility is not a good way to determine integrity!)

Cryptographic Hash Functions

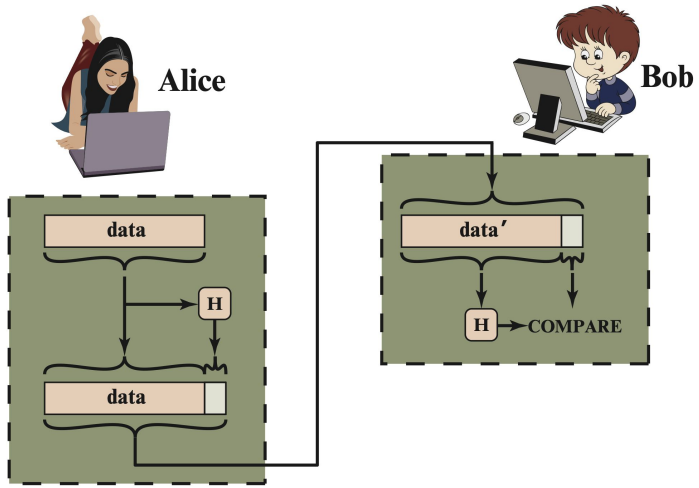
- Hash function that is needed for security applications
- It acts like one-way encryption
- **Length field:** a security measure to increase the difficulty for an attacker to produce h(a)= h(b), a ≠ b



P, L = padding plus length field

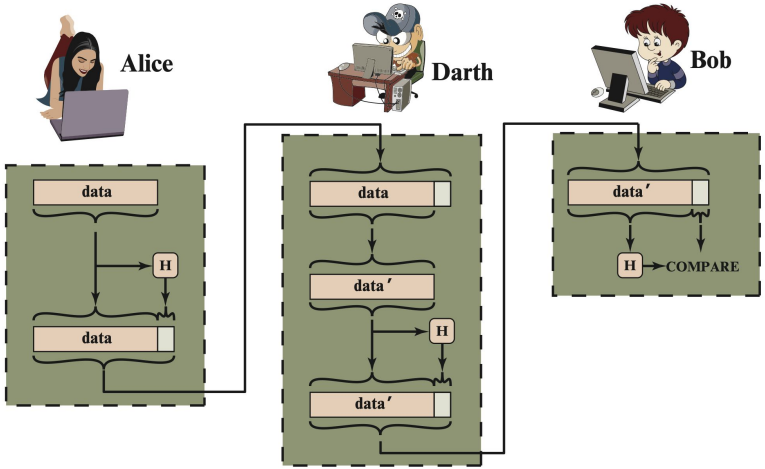
Applications

- Message authentication
- Verifies the integrity of the message
- Assures data received are exactly as sent
- Called ‘message digest’
- How does it works?
- Sender sends message m and hash h(m) to receiver
- Receiver uses the same hash function h to compute h(m’) and compare if h(m)= h(m’) where m′ is the received message



Applications

- Consider a Person-in-the-Middle attack
- This is the problem of origin authentication - it is closely related to the integrity
- **Solution:** Message Authentication Codes (MAC) - Also known as the keyed hash function
- Two parties share a common key
- Attacker can alter the message, but cannot alter the MAC value without knowing the MAC key
- We will discuss this later.



Applications

- **Digital Signature**
  - Hash value of the message is encrypted with a user's **Private Key (PR)**
  - Anyone who knows the user's **Public Key (PK)** can verify the message
  - Attack needs to know the user's **Private Key (PR)** to alter the message
  - We will discuss this later.
- **Others**
  - One-way password file
  - Intrusion detection
  - Virus detection
  - Construct a pseudorandom function (PRF) or Pseudorandom Number Generator (PRNG)

Cryptographic Hash Functions

- **Problem:** all hash functions have fixed-length output. Hence, every hash function H has collisions, i.e., H (a) = H (b), a ≠ b.
- We want to make finding them **hard**.
- Formally, we need the following to achieve this and prevent the mentioned attacks:
  - Preimage resistance
  - Second pre-image resistance
  - Collision resistance

2. Security Requirements of Cryptographic Hash Functions

Preimage Resistance

- **Goal**: If the attacker knows an output hash value, we don't want them to find an input value that would produce this output value.

Preimage Resistance - Given a randomly chosen y from H 's range of output values, it is computationally infeasible to find an x such that H (x) = y.

- Note that it is not impossible: can always brute-force. But computational infeasibility makes it much too hard in practice.

Second Preimage Resistance

- **Goal**: If the attacker is given an input value, they cannot find another input value that would have the same output hash value.

Second Preimage Resistance - Given a randomly chosen x , it is computationally infeasible to find an x' , $x' \neq x$ such that $H(x) = H(x')$.

Collision Resistance

- **Goal**: We want our attacker to be unable to find any two input values that have the same output hash value.

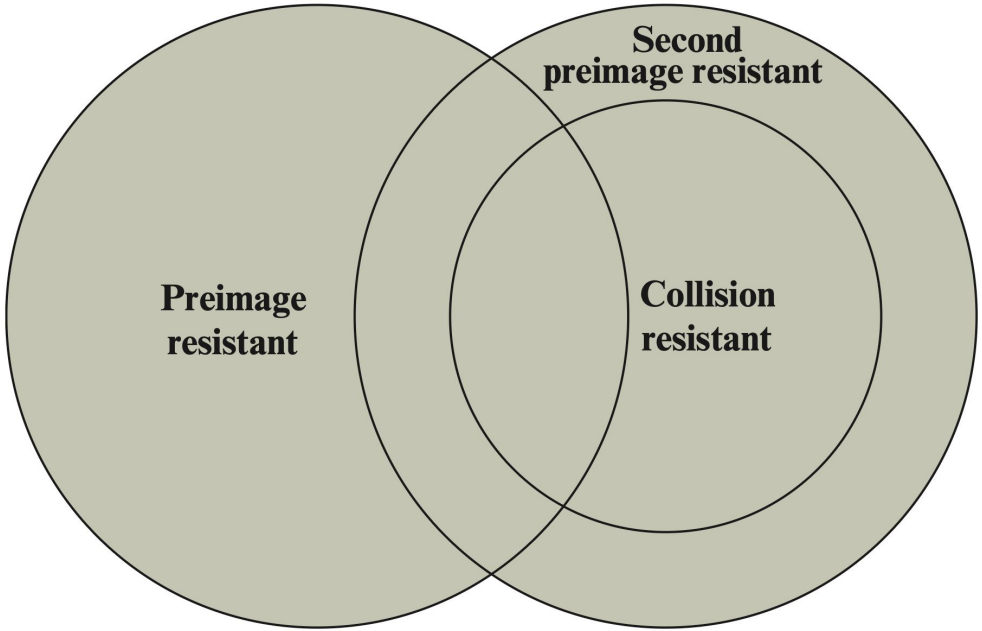
Collision Resistance - It is computationally infeasible to find any pair (x , x'), $x \neq x'$ such that $H(x) = H(x')$.

Collision resistance implies second-preimage resistance (but not preimage resistance).

Successful second-preimage attack implies successful collision attack.

- Successful collision can sometimes be all the attacker needs.
- Collisions are surprisingly a lot easier than you might imagine: Birthday attack

Hash Function Properties: Relationships



Collision resistance implies second preimage resistance - Any function that is hard to find collisions in is also hard to find a second input matching a given hash, but the reverse does not always hold

Preimage resistance stands apart - A function can be preimage resistant without being collision or second preimage resistant, and vice versa; no strict implication runs between them

Collision resistance is the strongest of the two related properties — if an attacker can break second preimage resistance (find a different input with the same hash as a given one), they have also broken collision resistance; meaning collision resistance is harder to satisfy and harder to attack

Why Does the Output Length Matter?

- Hash functions take inputs of arbitrary length and map them to a fixed-size output. Since the input space is infinite but the output space is finite, collisions are not just possible, they are mathematically guaranteed to exist.

- The question is therefore not whether collisions exist, but how hard they are to find.

- Collision resistance is the hardest property to satisfy but the easiest to attack. The attacker has maximum freedom, needing only any two inputs that collide, with no specific target. This is fundamentally easier than second preimage resistance, where the attacker is constrained to match a specific given input.

- **This raises a key question:** How many random attempts does it actually take before a collision becomes likely?

The Birthday Attack

Birthday Problem Definition - If there are 365 days in a year, how many people must be in a room before there is a 50% chance that two of them share the same birthday?" Answer: Just 23. This is far fewer than most people intuitively expect.

Key Observation:

The surprising result comes from the fact that we are not asking whether someone matches a specific birthday. We are simply asking whether any two people match each other. This gives the attacker vastly more opportunities to find a match.

Sketchy proof:

We add people one by one and the track the probability that all birthdays remain unique

1st person: No conflict possible → probability = **365/365 = 1**
2nd person: No conflict possible → probability = **364/365 = 0.99**
....

After k people P(no match) = **1.(364/365) .... ((365-k+1)/365)**

Solving for 50%

k	P(no match)
20	58.9%
22	52.4%
23	49.3%
25	43.1%

Birthday Attack in the Context of Hashes

- The birthday problem gives a surprising answer to hash collisions: you only need roughly √**(total possibilities)** attempts to have a 50% chance of finding a collision. This is far fewer than intuition suggests.

- **The 50% threshold is the standard benchmark in cryptanalysis:** If an attacker has better-than-coin-flip odds of succeeding, the function is considered practically broken.

- For a hash with n-bit output there are **2^n** possible hash values, so an attacker needs only **2^(n/2)** attempts. For example, at 64-bit output that is just 2^{32}, which is trivially achievable on modern hardware.

- This is why output length is a hard security requirement. Currently the recommendation is ≥160 bits. Here, the attacker needs 2^{80} attempts, which remains computationally infeasible today

Pseudo Randomness

- Pseudo randomness is not traditionally listed as a requirement of the cryptographic hash functions, but it's implied.
- Cryptographic hash functions are used for key derivation and pseudorandom number generation.
- Three resistant properties depend on the randomness of the output.
- Let's verify if a hash function produces pseudorandom output.

```
sh-3.2# echo "CYBERSECURITY ENGINEERING"|sha256sum
605d08ba60312b5e8b79105bc4f31ee8c269b956cada4821c9e22876bab917e5
```

```
sh-3.2# echo "CYBERSECURITY ENGINEERING"|md5sum
ca00a2f980e354eb1ff97c4305aaed2e
```

Cryptographic Hash Functions

- A function that fulfils preimage resistance, second-preimage resistance, and collision resistance is called a cryptographic hash function.

- We now have a (feasibility-conditioned) guarantee that the attacker cannot obtain plaintexts, and cannot easily find bogus inputs with the same hash values.

- A few more problems remain. We will revisit them again after learning the operation of some hash functions.

3. Secure Hash Algorithm (SHA)

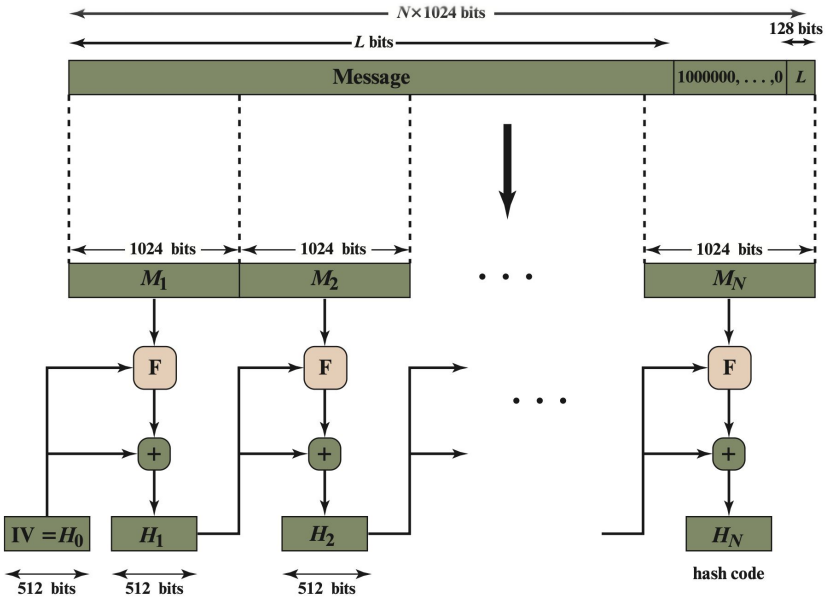
SHA (Secure Hash Algorithm)

- SHA-0, SHA-1 produce a hash value of 160 bits - They are no longer considered secure.
- **SHA-2 Family:** NIST then introduces SHA-224, SHA-256, SHA-384, SHA-512, also known as SHA-2 in 2001.
- **SHA-3 Family:** We also have SHA-3, the latest member of SHA family, released by NIST in 2015: SHA3-224, SHA3-256, SHA3-384, SHA3-512.
- We will discuss SHA-2 and SHA-3.

SHA-2 Design

SHA-512

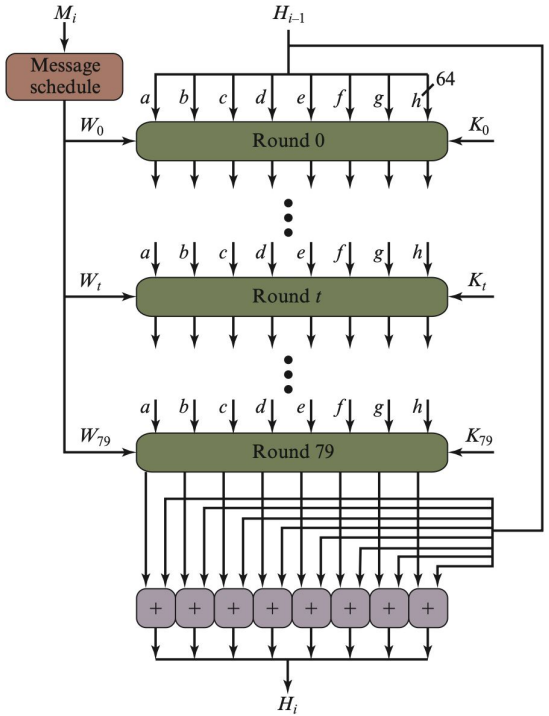
- Take input message with a maximum length of less than 2^{128} bits and output 512-bit message digest. Input is processed in 1024-bit blocks.
- The original message of L bits is padded with a single 1 bit followed by enough 0 bits so the total length is congruent to 896 mod 1024, leaving room at the end.
- A 128-bit encoding of the original length L is appended, bringing the padded message to an exact multiple of 1024 bits.
- The padded message is split into N blocks of 1024 bits each: M_1, M_2, …, M□. These are the "plaintext blocks" feeding the compression chain.
- H_0 (IV) is a fixed, publicly-specified 512-bit constant (for SHA-512) defined in the standard. It is not secret and not random.



SHA-2 Design

SHA-512 - What happens inside F?

- F unpacks H_{i-1} into eight 64-bit working registers a–h, which are transformed across 80 rounds
- Message schedule expands the 1024-bit block M_i into 80 round words W_0- W_{79}, analogous to AES key schedule turning one key into many round keys
- Each round mixes in one W□ and one public constant K□; six of the eight registers just shift along, so only two new values are computed per round.
- Davies–Meyer feedforward: after round 79, the final a–h are added mod 2^{64} back to the original H_{i-1} (the long bypass lines at the right of the figure).
- The result of those eight additions is H_i, passed to the next block



SHA-2 Applications

Applications:

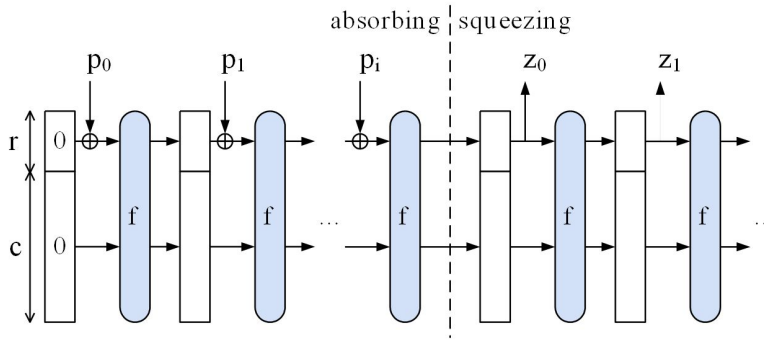
- Widely implemented in security applications and protocols (TLS/SSL, SSH, IPSec,…etc)
- Validates and signs digital certificates and documents
- Verifies transactions in cryptocurrencies (e.g., Bitcoin uses SHA-256)

Algorithm	Message Size	Block Size	Word Size	Message Digest Size
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Note: All sizes are measured in bits.

SHA-3

- **Single unified state** of r + c bits is passed through permutation f. Unlike SHA-2's eight working registers, there is one big state that gets repeatedly permuted
- **Absorbing phase**: each message block XORs into the top r bits of the state, then f scrambles the entire state. The bottom c bits are never directly touched by the message input, protecting the security margin
- **Squeezing phase**: output is read from the top r bits; f is applied between each read. You can squeeze out as many bits as needed, making SHA-3 naturally an extendable output function
- **Rate r vs capacity c is a tunable tradeoff**: larger r means faster hashing (more bits absorbed per round) but smaller c means lower security. Different SHA-3 variants just change this split
- **No length-extension vulnerability**: in SHA-2 an attacker who knows H(m) can compute H(m ‖ extra) without knowing m — the sponge structure eliminates this because the hidden c bits make the full state unrecoverable from the output



4. Message Authentication Codes

Message Authentication Codes

- Message Authentication Codes are special tags that can be constructed if Alice and Bob already share a secret key.
- Idea: create a MAC **t,** such that the attacker cannot find a second, correct MAC that is valid for a different message.
- Send **m** together with **t**.
- What must this MAC function be like?

Why MACs when we already have hashes?

Hashes are keyless. Anyone, including an attacker, can compute a valid hash for a forged message, so they prove the message wasn't corrupted but can't prove who sent it. A MAC fixes this by incorporating a secret key into the computation. Think of it as a keyed hash.

Message Authentication Codes

Requirements

- MAC function must have all resistance properties of a cryptographic hash function.
- Must also be computationally infeasible to predict a correct MAC, **t′** for a message **m′ ≠ m** even when allowed to know any other combination of **(m, t)**.

Construction

- A cryptographic hash function takes care of the first requirement. Mixing in a secret **s** will address the second requirement if the resistance properties hold.

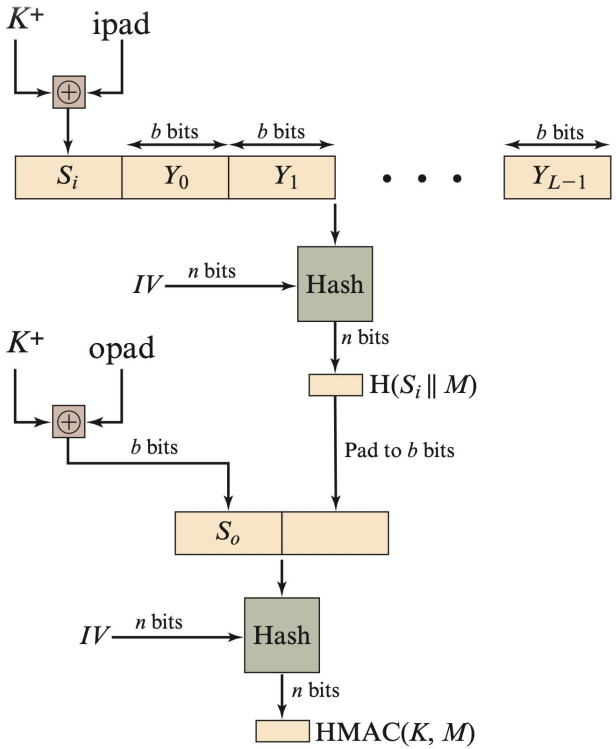
4a. Hash-based Message Authentication Code (HMAC)

Construction

$$\mathrm{HMAC}(k, m) = H((k \oplus \mathrm{opad}) \,//\, H((k \oplus \mathrm{ipad}) \,//\, m))$$

- H can be any cryptographic hash function: SHA-3, SHA-2, …
- ipad and opad are constant bit strings (publicly known)
- They are required for the security proof of HMAC to hold (precise values matter little, however)
- HMAC is probably the most common form of MACs
- Other forms of MACs exist, even without hashing - but we do not discuss them here.

HMAC Structure



Security of HMAC

- Depends in some way on the cryptographic strength of the underlying hash function and key size
- Appeal of HMAC is that its designers have been able to prove an exact relationship between the strength of the embedded hash function and the strength of HMAC
- **Question**: Can we use MD5 for HMAC?
  - Answer: **Yes**. Because:
  - The attacker does not know the secret key K, thus they must observe sequences of messages generated by HMAC online
  - 128-bit hash code length $\rightarrow 2^{64}$ observed blocks generated using the same key
  - On 1-Gbps link, attacker would need to observe a continuous stream of messages with no change in key for about 150,000 years to succeed.
  - If speed is a concern, we can use HMAC-MD5 over HMAC-SHA-1

Applications of HMAC

- HMAC is widely used wherever two parties share a secret key and need to verify both the authenticity and integrity of a message. i.e., not just that the message arrived uncorrupted, but that it came from a trusted sender

- **API Request Signing (e.g. AWS Signature V4)**
  - Client signs the full request (URL, headers, body, timestamp) with HMAC using a shared secret
  - Server recomputes the signature independently. A mismatch means the request was tampered with or replayed
  - Provides integrity and replay protection on top of standard authentication
  - **Note:** HMAC alone does not guarantee confidentiality. The message is still readable. In practice, this all happens over an encrypted TLS connection, which handles confidentiality separately.

- Other applications include JSON Web Tokens (JWT) signing and One-Time Passwords (HOTP/TOTP) generation.

4b. Authenticated Encryption (AE)

- So far we've treated confidentiality (encryption) and authenticity (MAC) as separate. Combining them correctly is tricky and error-prone
- AE and AEAD solve this by integrating both into a single, unified operation. This eliminates the risk of error, misusing or misordering the two primitives

Authenticated Encryption (AE) - AE provides confidentiality and integrity/authentication in a single primitive. The security of the construction depends on how tightly the two are coupled — ideally, no plaintext is released until integrity is verified.

Authenticated Encryption with Associated Data (AEAD) - AEAD extends AE by allowing additional data (associated data) that is authenticated but not encrypted. This data is sent in plaintext but included in the integrity check.

Authenticated Encryption (AE) - Approaches

- **Four Approaches:**
  - Hashing followed by encryption (Hash-then-Encrypt):
    - Send **$Enc_k(m, H(m))$**
  - Authentication followed by encryption (MAC-then-Encrypt)*:*
    - Send **$Enc_{k1}(m, MAC_{k2}(m))$**
    - Used in SSL/TLS
  - Encryption followed by authentication (Encrypt-then-MAC)*:*
    - Send **$Enc_{k1}(m), MAC_{k2}(Enc_{k1}(m))$**
    - Used in IPSec protocol/ Considered the most strongest
  - Independently encrypt and authenticate (MAC-and-Encrypt)*:*
    - Send **$Enc_{k1}(m), MAC_{k2}(m)$**
    - Used in SSH protocol

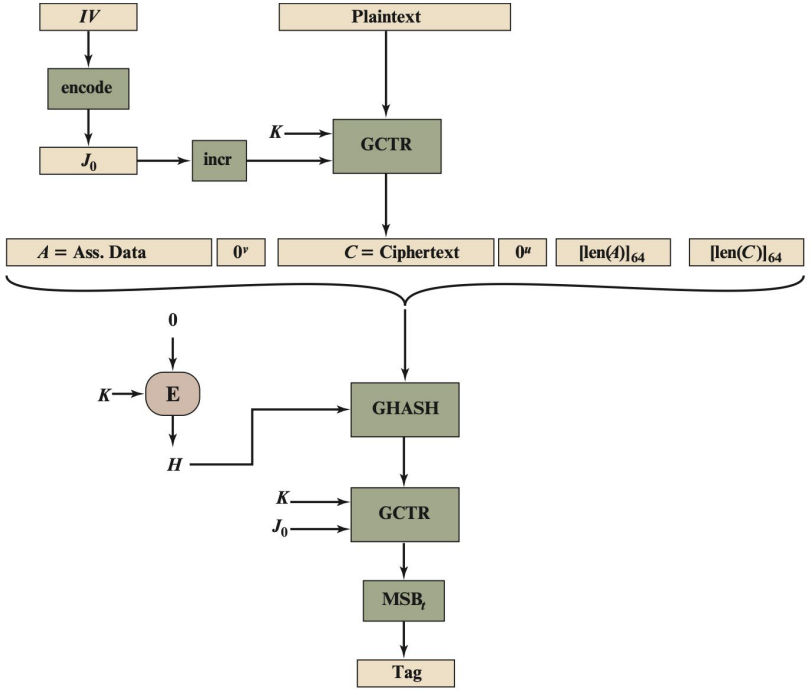
4c. Authenticated Encryption with Associated Data (AEAD)

- Extends AE by also authenticating associated data data that must remain readable (e.g. headers, metadata) but still needs tamper protection
- The MAC tag computation and encryption can be performed in parallel, making AEAD schemes efficient
- Defined for both block ciphers and stream ciphers
- Two prominent constructions:
  - **GCM (Galois/Counter Mode)** - Currently the strongest and most widely used block cipher AEAD mode; highly efficient due to parallelism. However, it is very brittle in implementation. Nonce reuse, for example, can completely break security. Always use a well-vetted library rather than implementing it yourself.
  - **CCM (Counter with CBC-MAC)** - Computes CBC-MAC over the message to generate a tag, then encrypts everything using CTR mode. This makes it a MAC-then-Encrypt construction. Less efficient than GCM (sequential), but widely used in constrained environments like IoT and WiFi (802.11).

Example AEAD - Galois Counter Mode (GCM)

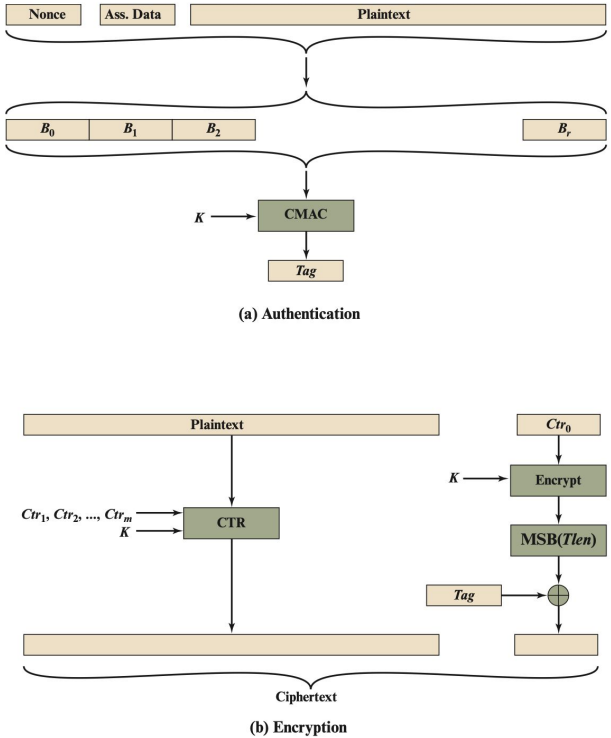
- Designed to be parallelizable so that it can provide high throughput with low cost and low latency. Follows the Encrypt-then-MAC paradigm.
  - Message is encrypted in the variant of CTR mode
  - Resulting ciphertext is multiplied with key material and message length information over GF (2^{128}) to generate the authenticator tag
  - The standard also specifies a mode of operation that supplies the MAC only, known as Galois MAC

- Makes use of two functions:
  - **GHASH** - a keyed hash function
  - **GCTR** - CTR mode with the counters determined by simple increment by one operation



Example AEAD - Counter with Cipher Block Chaining - MAC

- Standardized by NIST to support the security requirements of IEEE 802.11 WiFi networks.
- A carefully engineered variation of MAC-and-Encrypt. i.e., It computes CMAC over the message first to generate a tag, then encrypts both the message and tag using CTR mode.
- Unlike naive MAC-and-Encrypt, CCM is still considered secure because the construction is tightly specified. But it is sequential (MAC must complete before encryption begins), making it less efficient than GCM.
- A single key K is used for both encryption and MAC. This is safe here for the same reason as GCM: the two uses are carefully separated by the design and accounted for in the security proof.

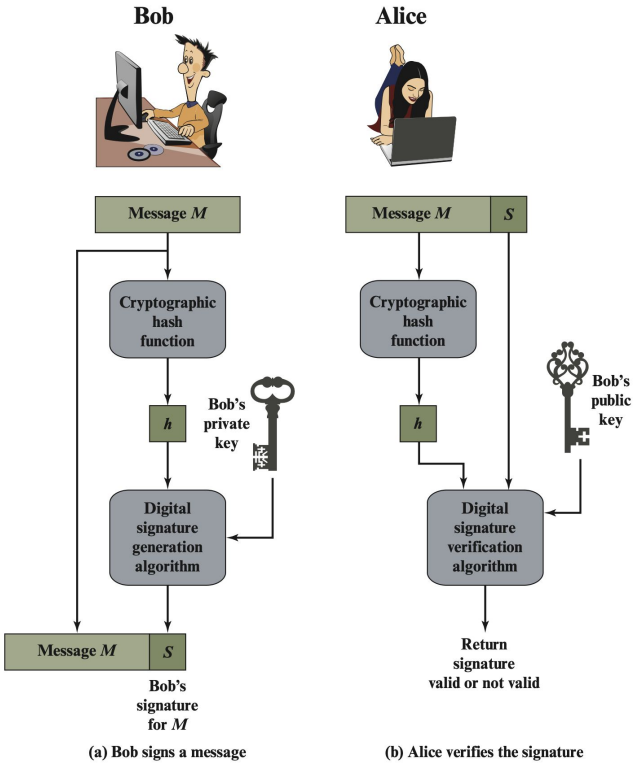


5. Digital Signatures

Overview of Digital Signatures - The Need

- Hashes give us integrity. But no notion of who sent the message. MACs give us integrity + authenticity. But rely on a shared secret key, meaning both Alice and Bob must already have agreed on a key
- This creates a fundamental limitation: MAC ….
  - Cannot prove to a third party that Alice sent the message - Bob could have computed the same MAC himself, so a judge or external verifier has no way to distinguish
  - Do not scale - If Alice wants to communicate securely with 1000 people, she needs 1000 different keys
  - No non-repudiation - Alice can later deny sending the message, and there's no way to prove otherwise
- Digital signatures solve this by using asymmetric (public key) cryptography - Alice signs with her private key, anyone can verify with her public key
  - No shared secret needed
  - A third party can verify the signature independently
  - Alice cannot deny signing. Providing non-repudiation

Digital Signatures - Basic Operation



Source: Cryptography and Network Security (Seventh Edition - William Stallings)

Digital Signatures - Basic Operation

Bob signs the hash of the message m using their private key $k_{d,B}$:

$$Sig_B(m) = Enc_{k_{d,B}}(h(m))$$

Bob encrypts the message with Alice's public key $k_{e,A}$ and appends the signature:

$$c = Enc_{k_{e,A}}(m), Enc_{k_{d,B}}(h(m))$$

Shorthand notation:

$$B \rightarrow A : c, Sig_B(m)$$

Alice decrypts the ciphertext using their private key $k_{d,A}$ to recover m:

$$m = Dec_{k_{d,A}}(c)$$

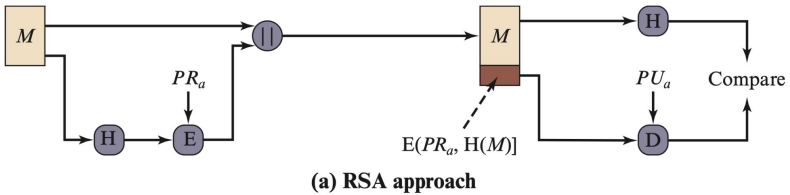
Alice then decrypts the signature using Bob's public key $k_{e,B}$ and verifies $h(m)$:

$$h(m) \stackrel{?}{=} Dec_{k_{e,B}}(Sig_B(m))$$

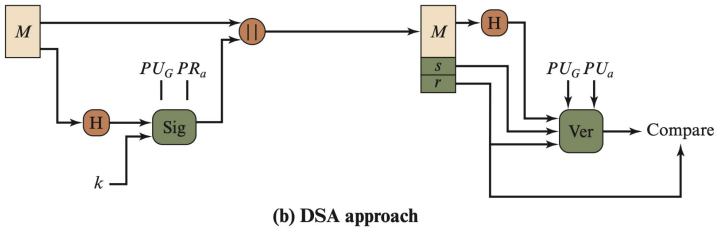
Caveat: This only works if the verifier (Alice) can be certain that the public key she has actually belongs to Bob. A forged or substituted public key breaks all guarantees. This is the problem of public key authentication, and there is a way to solve it (PKI). We learn this later under Digital Certificates.

Digital Signatures - Approaches

- Two main approaches for constructing digital signatures. Both approaches use SHA for hashing
  - (a) RSA Approach
    - The message m is hashed using SHA, producing h(m)
    - The hash is encrypted with the sender's private key PR_a to produce the signature $E(PR_a,h(m))$
    - The receiver verifies by decrypting the signature with the sender's public key PU_a and comparing with their own computed h(m)
  - Recall: RSA is a general-purpose public key scheme. It can also be used for encryption and key exchange.
  - (b) DSA (Digital Signature Algorithm)
    - A dedicated signature-only scheme — cannot be used for encryption or key exchange, unlike RSA. Derived from the ElGamal signatures
    - Uses a global public key PU_G shared across all users (system-wide parameters), plus the sender's private key PR_a and a random per-message k
    - Produces a signature consisting of two components (s,r), verified using PU_G, PU_a
    - NIST standardized DSA as a signature-only primitive, intentionally excluding the encryption capability that ElGamal supports



(a) RSA approach



(b) DSA approach

Recap

- **We discussed:**
  - Cryptographic hash function basics
  - Security requirements for cryptographic hash functions
  - Secure Hash Algorithm (SHA-2, SHA-3)
  - Message Authentication Codes and HMAC
  - AE/AEAD (CCM and GCM block mode)
  - Digital Signatures (RSA, DSA)

Digital Signature Algorithm

Let p be prime, q a prime divisor of (p−1), and g, an element of order q in $\mathbb{Z}_p^*$

Domain parameters PU_G= (p, q, g) shared by all users.

Alice

- Choose random value a < p
- Compute $X = g^a \mod p$
- Send X to Bob
- Compute $k = Y^a \mod p$

Bob

- Choose random value b < p
- Compute $Y = g^b \mod p$
- Send Y to Alice
- Compute $k = X^b \mod p$